Exploring GA Techniques and Parameters for Solving Hardest 1-0 Knapsack Instances

Ashwin Padmanabhan

September 29, 2019

The section below "Logic Review" is a brief explanation of the logic and assumptions used to implement the GA.

1 Logic Review:

1.1 Overall Run Parameters:

- **Representation**: A binary 1-D array of length equal to number of items available to place inside Knapsack. 0/1 representing item is/is not added in the Knapsack.
- **Initial Population**: Start with 40 valid individuals chosen randomly. (Section 1.2)
- **Progeny progression**: Two parents produce two children.
- **Number of generations**: Most cases are run for 700 generations. This is chosen because this seems to be a good balance between having enough generations for convergence and computational cost for each generation.

1.2 Initialize Population:

```
1  def initializepopulation( instanceinfo, inpopulationsize, weight,
2                            knapsackcapacity, repmode = "binary"):
3      if repmode == 'binary':
4          initialpopulation =[]
5          while( len(initialpopulation)< inpopulationsize ):
6              generateindividual = bernoulli.rvs(0.01,
7                                  size = int(instanceinfo[2]) )
8              isfeasible = isFeasibleSolution(generateindividual,
9                                  weight, knapsackcapacity)
10             if ((isfeasible) and (listcompare(initialpopulation,
11                                      generateindividual))):
12                 initialpopulation.append(generateindividual)
13         return(initialpopulation)
```

Initial population selection is done using random Bernoulli sampling. **initializepopulation()** returns, "initialpopulation", a (Initial population Size x len(genotype)) sized list. Population is initialized using a Bernoulli trial value on each position of the genotype, "initial population size" number of times. Within the loop (line 5), line 8 checks if the genotype produced satisfies the capacity constraint of the Knapsack and is appended into the population only if it does. This is the constraint handling used here. Individuals (children, mostly) are checked on whether they satisfy the Knapsack capacity constraint and are added to the population only if they do. The probability of the Bernoulli success is manually set to 0.01 using a trial-error approach, since this produces best reasonable initial individuals to get us started given the object weight and knapsack capacity (especially for instance 2).

1.3 Selection:

```
1  def selection(population, population1, values, mode= "tournament"):
2      if mode == "tournament":
3          parents = []
4          for _ in range(2):
5              candidateparent = [random.randrange(0,len(population))
6                                  for _ in range(2)]
```

```
7               parent = np.argmax(
8                   [population1[candidateparent[0]][1],
9                       population1[candidateparent[1]][1]])
10              parent = population[candidateparent[parent]]
11              parents.append(parent)
12          return(parents)
```

A tournament selection is implemented here. Line 5 above produces two random indices between **[0,len(population))** and a tournament takes place to select the parent with a higher fitness value which then goes on to be one of the parent.

1.4 Recombination and Mutation:

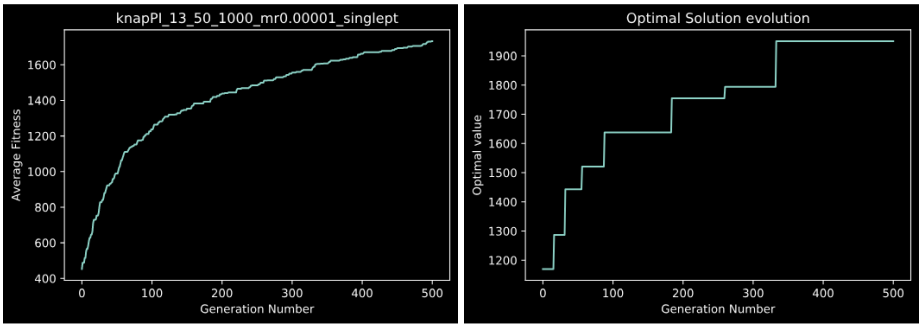
```
1  #Recombination and Mutation.
2           for i in range(len(parentpair)):
3               children = []
4               generatecrossoverflag= bernoulli.rvs(crossoverrate, size = 1)
5               if generatecrossoverflag ==1:
6                   children= [np.concatenate((parentpair[i][0][0:crossoverpoint],
                        parentpair[i][1][crossoverpoint:int(instanceinfo[2])]), axis
                         =0),
7                                   np.concatenate((parentpair[i][1][0:
                                        crossoverpoint], parentpair[i][0][
                                        crossoverpoint: int(instanceinfo[2])]), axis
                                         = 0)]
8               else:
9                   children = [parentpair[i][0], parentpair[i][1]]
10              #Mutation:
11              children=mutate(children, mutationrate)
12              for j in range(len(children)):
13                  isfeasible = isFeasibleSolution(children[j], weights,
                        knapsackcapacity)
14                  if ((isfeasible) and listcompare(population, children[j])):
15                      population.append(children[j])
16                      population1.append([children[j], calcobj(children[j],
                            values)])
17                      presentindex = killindividual(population1)
18                      population.pop(presentindex)
19                      population1.pop(presentindex)
```

A mutation and crossover probability (or rate) is user assigned at the beginning of the run. Crossover does a 1-pt crossover to produce the genotype of the child. Once crossover is done, Mutation flips the value of a genotype position to it's compliment. Also please note that (following line 12) once children has been created they are appended into the population only if the same genotype does not exist already in the population (Boolean output function **listcompare()**) and if the genotype produces a valid phenotype (Boolean output function **isFeasible()**) (Weight occupied by items in the Knapsack). If it is infeasible, the child is not appended to the population. **killindividual()** returns index to removes the same amount of individuals as the number of children added to keep the overall population constant.

Mutation: Whenever a child is produced, mutation is done at every location of the genotype with an equal probability as entered by the mutation rate. For example, for a mutation rate of 10^{-5} every bit (position) of the genotype is mutated equal Bernoulli success rate of 10^{-5}. For question 3, mutation does not take place every time a child is created. It takes place with a probability given by the mutation rate (around 0.1), and whenever mutation is to be done, a random location from the genotype is sampled and inverted. The code above calls the fitness function **calcobj()** *only when* the child produced is not already present in the existing population. The overall times the fitness evalution function, calcobj(), is called: At the beginning when initial population is defined and when a unique child is produced. This is the reason why we do not need to make too many calls to the fitness function.

1.5 Evolution:

Below is a plot showing average fitness and optimal solution evolution for the instance in file, knapPI_13_50_1000.csv, running for 500 generations. Other evolution plots in submitted folder under "Evolution Plots".



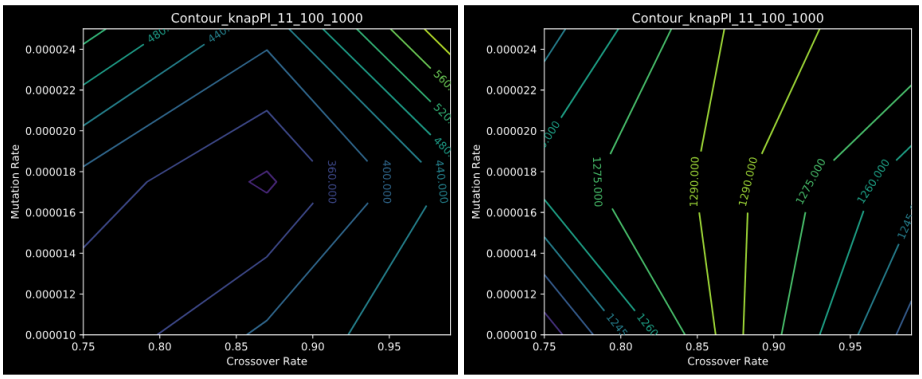
(a) Figure A (b) Figure B

Figure 1: (A) Average Fitness v Generation (B) Optimal Evolution

2 Impact of Mutation and Crossover:

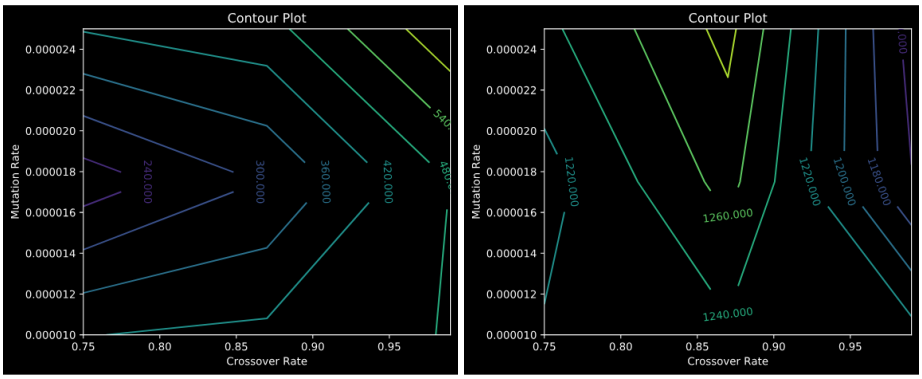
Study impact of the the balance between mutation and crossover (for both uniform and single point) on algorithm run time (number of function calls of the fitness function) and fitness.

To study the balance of crossover and mutation rates, we will do a contour plot for a range of cross-over and mutation probabilities, plotting contours for both number of fitness function calls and average fitness.
Contour plots for the six instances given are shown below. The GA is run for given set of repetitions (5 here, fig. 4 being an exception, which was run twice) to calculate the average fitness value at each of the crossover-mutation rate combinations.



(a) Number of fitness function calls (b) Average Fitness

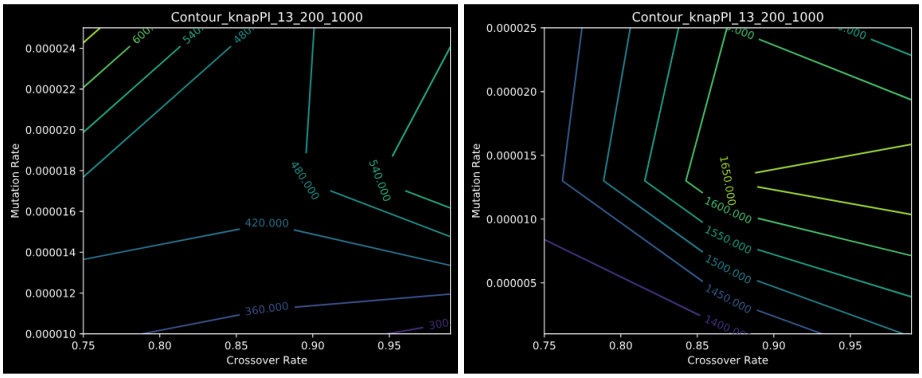
Figure 2: Contour plots for instance **knapPI_11_100_1000**: (A) Function calls (B) Average fitness



(a) Number of fitness function calls (b) Average Fitness

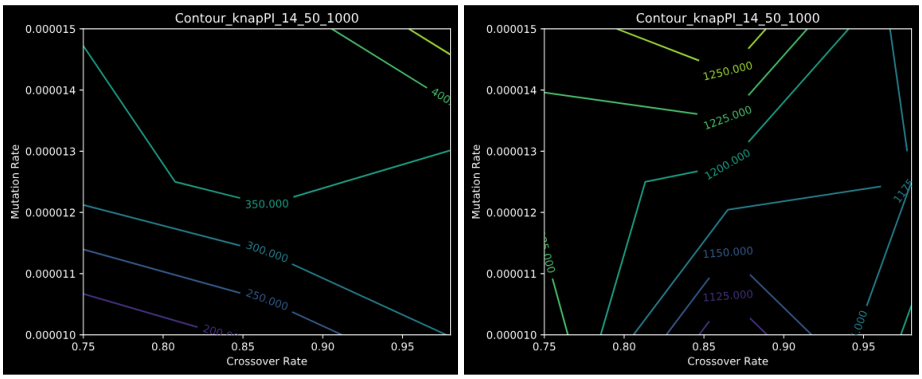
Figure 3: Contour plots for instance **knapPI_13_50_1000**: (A) Function calls (B) Average fitness

Comparing the contour plots, it appears that a cross-over probability of 0.85 and a mutation probability of 0.000018 seems to not only produce less number of function calls but provides a decent average optimal solution.



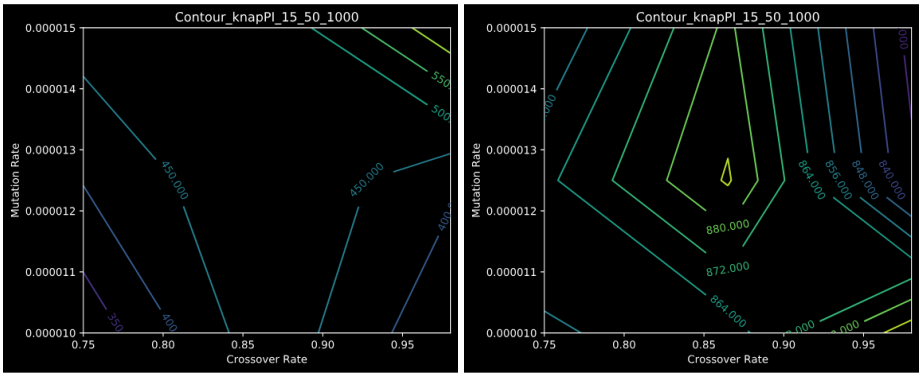
(a) Number of fitness function calls (b) Average Fitness

Figure 4: Contour plots for instance **knapPI_13_200_1000**: (A) Function calls (B) Average fitness



(a) Number of fitness function calls (b) Average Fitness

Figure 5: Contour plots for instance **knapPI_14_50_1000**: (A) Function calls (B) Average fitness



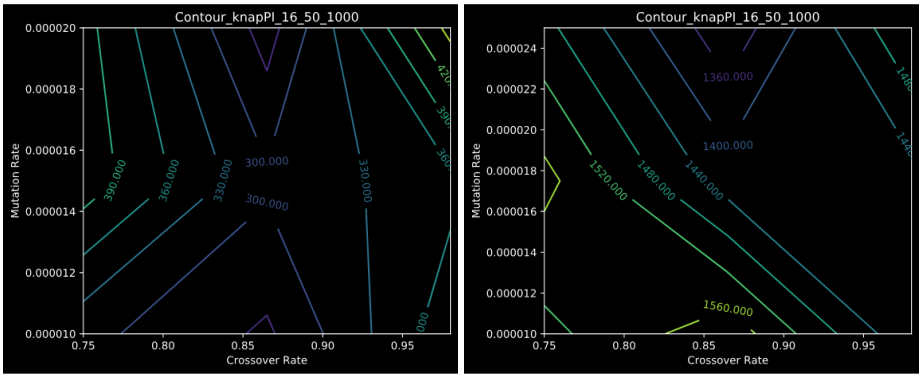
(a) Number of fitness function calls (b) Average Fitness

Figure 6: Contour plots for instance **knapPI_15_50_1000**: (A) Function calls (B) Average fitness

3 Comparison to Random Sampling:

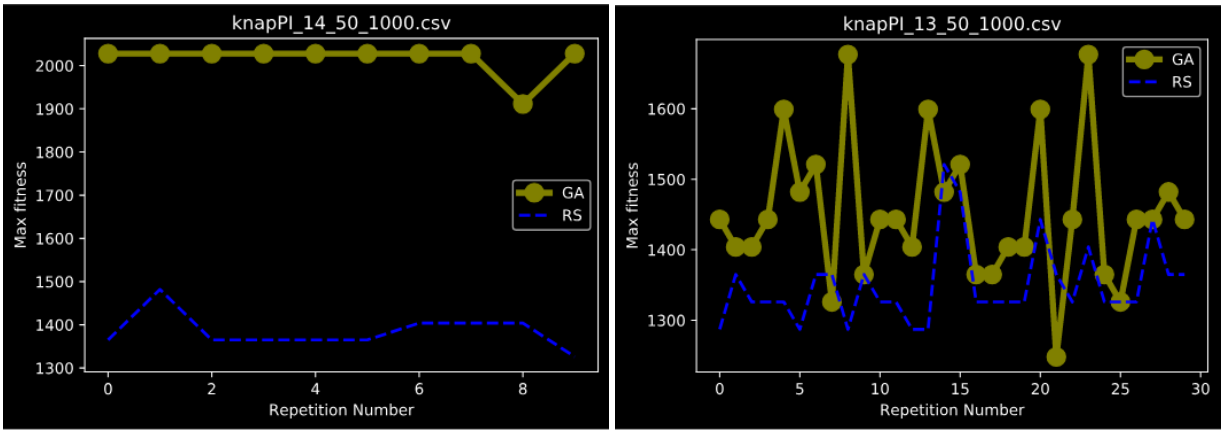
Compare the performance of the "best" parameters from part (a) to a random sampling approach for different population sizes, ensuring a fair comparison (equal total number of fitness function calls).

I have scripted my program is such a way that the number of function calls decides the population size for the Random Sampling (RS). The best solution of the RS data was compared with the resulting best solution of the GA. This comparison was repeated 20 times for 2 problem instances, **"knapPI_13_50_1000.csv"** and **"knapPI_14_50_1000.csv"**. In both cases GA performs significantly better than RS evidenced by the line graph and the box plot below. The number of fitness calls made at each run ranged from 250-400, which represents the population of the randomly sampled data at those runs.



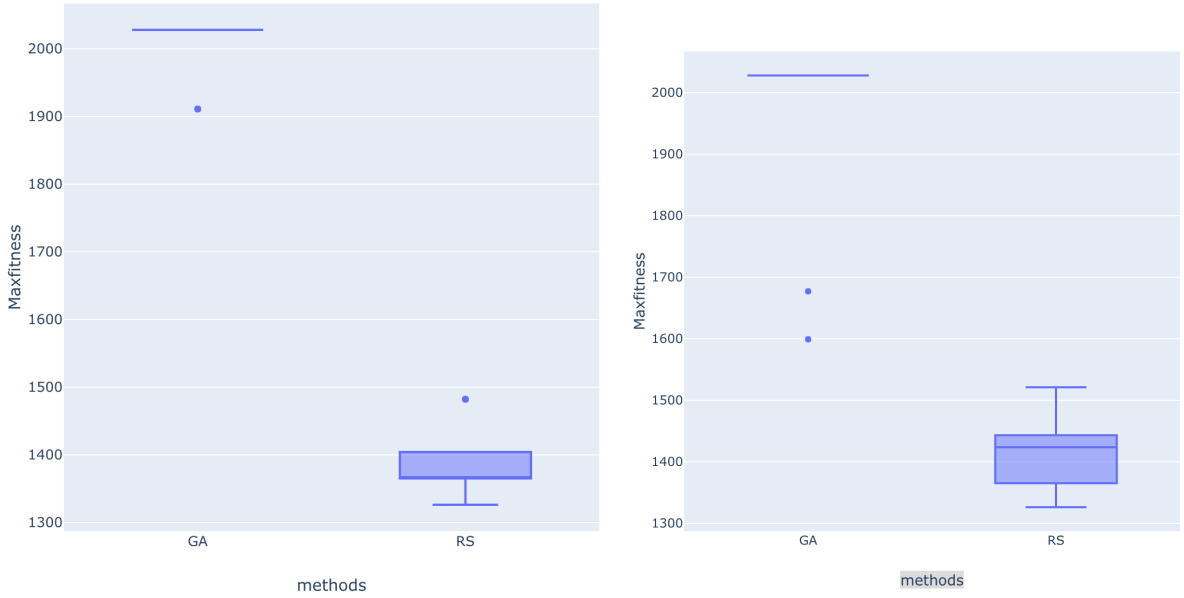
(a) Number of fitness function calls (b) Average Fitness

Figure 7: Contour plots for instance **knapPI_16_50_1000**: (A) Function calls (B) Average fitness



(a) For instance KnapPI14_50_1000 (b) For instance KnapPI13_50_1000 calls

Figure 8



(a) For instance KnapPI14_50_1000 (b) For instance KnapPI13_50_1000 calls

Figure 9

4 New mutation method:

I am modifying the way mutation is being done. Earlier, a Bernoulli trial was done for each position of the genotype for the child produced. Instead, I think it is cleaner to first ask question "Is this child a mutant?" which will give me a "Yes" with a mutation probability (say around 0.01) and then I randomly select a position within the genotype of the child and compliment the entry. I think this will be much more scalable with increasing genotypic lengths. I ran the GA for 50 repetitions, for 500 iterations, each time, using both methods and noted the average fitness of population in each case. Method 2 seems to do much better than method 1, in terms of average fitness, however, at the cost of higher fitness calls. (Histograms saved in the folder). I performed a T test for the means of the two fitness/function calls values as two independent samples assuming equal variance. I obtained a p-value of 0.00417 and 0.12518 for fitness and function calls respectively. It suggests that the difference in fitness is much more significant that the difference in function calls. I would use this method of mutation over the previous one.

The diagram below shows the distribution of function calls for the two methods.

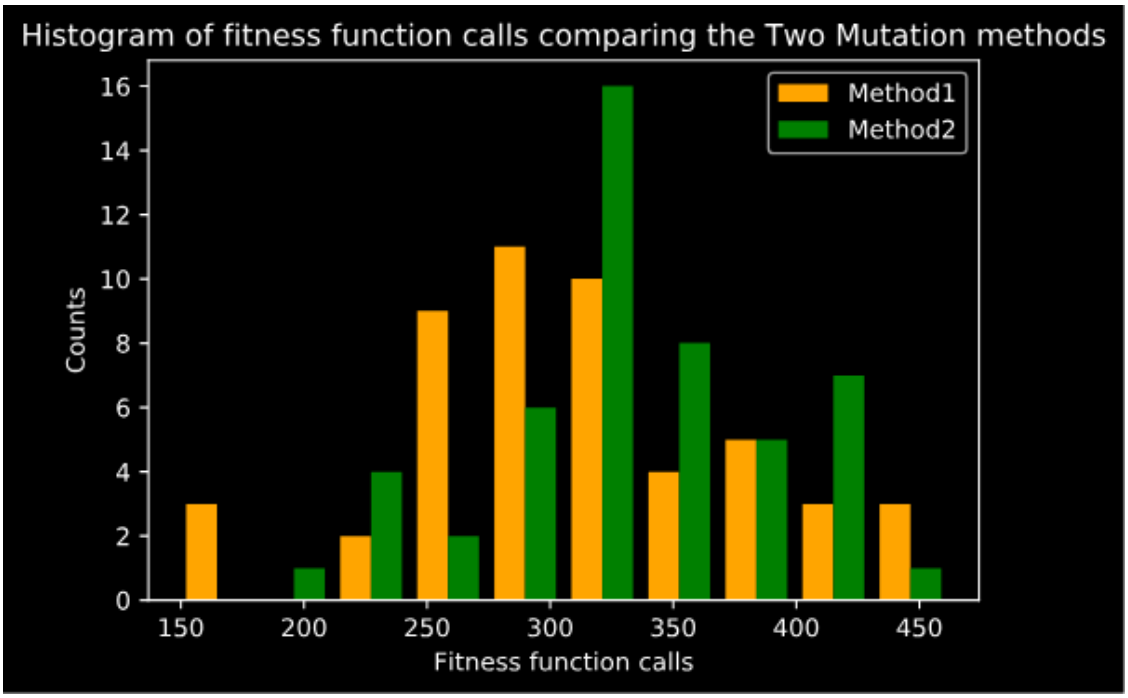
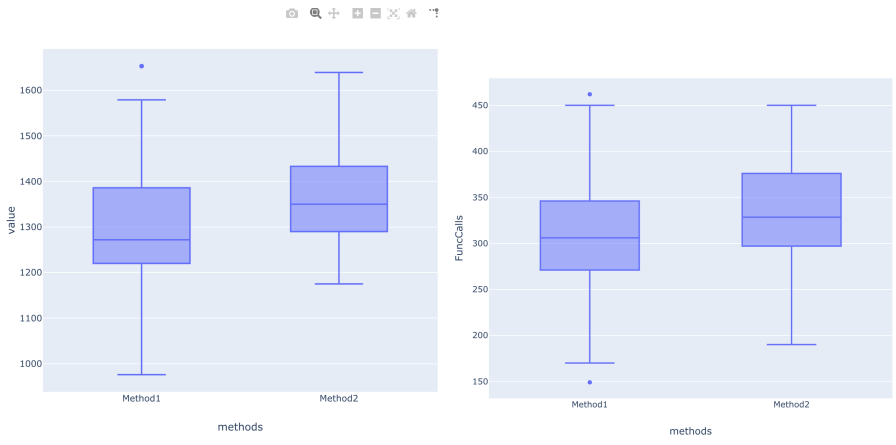


Figure 10



(a) Average Fitness (b) Number of fitness function calls

Figure 11: Different Methods of Mutation